Reducing Thread Divergence in GPU-based B&B Applied to the Flow-shop problem

Imen Chakroun [1], Ahcène Bendjoudi [2], Nouredine Melab[1]

[1]Université Lille 1 CNRS/LIFL, INRIA Lille Nord Europe
Cité scientifique - 59655, Villeneuve d'Ascq cedex - France
{imen.chakroun,nouredine.melab}@lifl.fr
[2]CEntre de Recherche sur l'Information Scientifique et Technique (CERIST)
Division Théorie et Ingénierie des Systèmes Informatiques DTISI,
3 rue des frères Aissou, 16030 Ben-Aknoun, Algiers, Algeria
{abendjoudi}@cerist.dz

Abstract. *In this paper,we propose a pioneering work on designing and programming B&B algorithms on GPU. To the best of our knowledge, no contribution has been proposed to raise such challenge. We focus on the parallel evaluation of the bounds for the Flow-shop scheduling problem. To deal with thread divergence caused by the bounding operation, we investigate two software based approaches called thread data reordering and branch refactoring. Experiments reported that parallel evaluation of bounds speeds up execution up to 54.5 times compared to a CPU version.*

Keywords: Branch and Bound, Data Parallelism, GPU Computing, Thread Divergence, Flow-shop Scheduling

1 Introduction

Solving to optimality large size combinatorial optimization problems using a Branch and Bound algorithm (B&B) is CPU time intensive. Although B&B allows to reduce considerably the exploration time using a bounding mechanism, the computation time remains significant and the use of parallelism to speed up the execution has become an attractive way out. Because of their tremendous computing power and remarkable cost efficiency, GPUs (Graphic Processing Units) have been recently revealed as a powerful way to achieve high performance on long-running scientific applications [9]. However, while several parallel B&B strategies based on large computer clusters and grids have been proposed in the litterature [7], to the best of our knowledge no contribution has been proposed for designing B&B algorithms on GPUs. Indeed, the efficient parallel B&B approaches proposed in the literature [2] do not immediately fit GPU architecture and have to be revisited.

B&B algorithms are characterized by four basic operations: branching, bounding, selection and elimination. For most combinatorial problems, bounding is a very time consuming operation. Indeed, a bounding function is used to compute the estimated optimal solution called *lower bound* of the problem being tackled.

For this reason, and in order to reach higher computing performance, we focus on a GPU based B&B algorithm using a parallel evaluation of the bounds. This parallel strategy is a node-based approach. It does not aim to modify the search trajectory, neither the dimension of the B&B tree nor its exploration. The main objective is to speed up the evaluation of the lower bounds associated to the sub-problems using a GPU CUDA-based computing without changing the semantics of the execution.

The design and programming paradigm proposed in CUDA is based on the Simple Program Multiple Data (SPMD) model. However, its execution model is Single Instruction Multiple Data (SIMD) which is well suited for regular functions (kernels) but represent a challenging issue for irregular computations. In this paper, we address such issue on the Flow-shop scheduling problem for which the bounding function is irregular leading to thread divergence. Indeed, if sub-problems evaluated in parallel by a warp of threads (32 threads in the G80 GPU model) are located at different levels of the search tree, the threads may diverge. This means that at a given time they execute different instruction flows. This behavior is due to the bounding function for the Flow-shop problem which is composed of several conditional instructions and loops that depend on the data associated to the sub-problem on which it is applied. We investigate two approaches called thread data reordering and branch refactoring to deal with thread divergence for the Flow-shop scheduling on GPU.

The remainder of the paper is organized as follows: in Section 2, we present the different B&B parallel existing models focusing on the parallel evaluation of bounds. We also highlight the issues and challenges to deal with the irregular nature of the bounding operation of the Flow-shop problem. In Section 3, we analyse the thread divergence scenarios for this problem, our case study. While in Section 4, we show how to reduce thread divergence using a judicious thread data remapping, we detail in Section 5 some software optimizations useful to get around control flow instructions. Finally, some perspectives of our work are proposed in Section 6.

2 GPU-based parallel B&B: issues and challenges

Solving exactly a combinatorial optimization problem consists in finding the solution having the optimal cost. For this purpose, the B&B algorithm is based on an implicit enumeration of all the solutions of the problem being solved. The space of potential solutions (search space) is explored by dynamically building a tree which root node represents the initial problem. The leaf nodes are the possible solutions and the internal nodes are subspaces of the total search space. The construction of such a tree and its exploration are performed using four operators: branching, bounding, selection and pruning. The bounding operation is used to compute the estimated optimal solution called “lower bound” of the problem being tackled. The pruning operation uses this bound to decide whether to prune the node or to continue its exploration. A selection or exploration strategy selects one node among all pending nodes according to defined priorities.

The priority of a node could be based on its depth in the B&B tree which leads to a depth-first exploration strategy. A priority based on the breadth of the node is called a breadth-first exploration. A best first selection strategy could also be used. It is based on the presumed capacity of the node to yield good solutions.

Thanks to the pruning operator, B&B allows to reduce considerably the computation time needed to explore the whole solution space. However, the exploration time remains significant and parallel processing is thus required. In [7], three parallel models are identified for B&B algorithms: (1) the parallel multi-parametric model (2), the parallel tree exploration, and (3) the parallel evaluation of the bounds. The model (1) consists in launching simultaneously several B&B processes. These processes differ by one or more operator(s), or have the same operators differently parameterized. The trees explored in this model are not necessarily the same. Model (2) consists in launching several B&B processes to explore simultaneously different paths of the same tree.

Unlike the two previous models, model (3) suppose the launching of only one B&B process. It does not assume to parallelize the whole B&B algorithm but only the bounding operator. Each calculation unit evaluates the bounds of a distinct pool of nodes. This approach perfectly suits GPU computing. In fact, bounding is often a very time consuming operation. In this paper, we focus on the design and implementation of a B&B algorithm on GPU based on the parallel evaluation of the lower bounds.

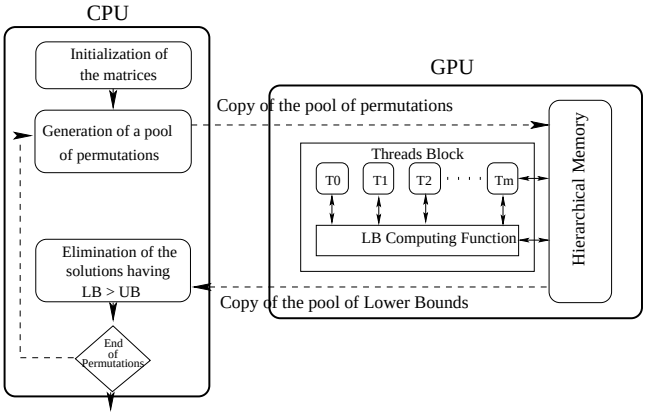

Fig. 1. GPU-based evaluation of bounds

As illustrated in Figure 1, the idea is to generate a pool of subnodes on CPU using the branching operator and to send it to GPU where each thread handles one node. The subnodes are then evaluated in parallel, the resulting lower bounds are moved back to CPU where the remaining selection and elimination operators are applied.

Using Graphics Processing Units have become increasingly popular in High-Performance Computing. A large number of optimizations have been proposed to improve the performance of GPU programs. The majority of these optimizations target the GPU memory hierarchy by adjusting the pattern of accesses to the device memory [9]. In contrast, there has been less work on optimizations that tackle another fundamental aspect of GPU performance, namely its SIMD execution model. This is the main challenge we are facing in our work. When a GPU application runs, each GPU multiprocessor is given one or more thread block(s) to execute. Those threads are partitioned into warps [1] that get scheduled for execution. At any clock cycle, each processor of the multiprocessor selects a half warp that is ready to execute the same instruction on different data. The GPU SIMD model assumes that a warp executes one common instruction at a time. Consequently, full efficiency is realized when all 32 threads of a warp agree on their execution path. However, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken. Threads that are not on that path are disabled, and when all paths complete, the threads converge back to the same execution path. This phenomenon is called *thread divergence* and often causes serious performance degradations.

The parallel evaluation of bounds is a node-based parallelism. This feature implies an irregular computation depending on the data of each node. Irregularities calculation are reflected in several flow control instructions that would conduct to different behaviors. As we explained before, such data-dependent conditional branches are the main cause of thread divergence. In the following section, we discuss such conditional instruction we encounter in the lower bound of the Flow-shop permutation problem.

3 Thread divergence in the Flow-shop lower bound

The permutation Flow-shop problem is a very known NP-hard combinatorial optimization problem. It can be formulated as a set of N jobs J_1, J_2..J_N to be scheduled in the same order on M machines. The machines are critical resources as each machine can not be simultaneously assigned to two jobs. Each job J_i is composed of M consecutive tasks $ti_1..ti_M$, where t_{ij} designates the j^{th} task of the job J_i requiring the machine M_j. To each task t_{ij} is associated a processing time p_{ij}. The goal is to find a permutation schedule that minimizes the total processing time called makespan.

The effectiveness of B&B algorithms resides in the use of a good estimation (lower bound for the maximization problem) of the optimal solution. In that purpose we use the most known lower bound for the permutation Flow-shop problem with M machines; the one proposed by Lageweg et al. [6] with $O(M^2NlogN)$ complexity. This bound is based mainly on Johnson's theorem [5], which provides a procedure for finding an optimal solution with 2 machines. Johnson algorithm assumes to assign jobs at the beginning or at the end of the schedule depending of the processing time of that job.

[1] We assume using the G80 model in which a warp is a pool of 32 threads

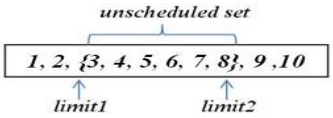

Fig. 2. Representation of the thread input

Starting from this principle of Johnson's algorithm, we designed a thread input as a set of unscheduled jobs, an index representing the start of the range of unscheduled jobs namely LIMIT1 and an index addressing the end of the range of the unscheduled jobs namely LIMIT2 (see Figure 2). Each thread would pick one of the unscheduled jobs, schedule it and calculate the corresponding makespan.

The Flow-shop permutation lower bound we adopted clearly provides a good estimation of the cost of a solution. However, its implementation on GPU perfectly echoes the thread divergence. Actually, it counts almost a dozen of control instructions namely "if" and "for". The example above shows a piece of our code that exhibits thread divergence.

```
int thread_idx = blockIdx.x * blockDim.x + threadIdx.x;

1. if( pool[thread_idx].limit1 != 0 )
      time = TimeMachines[1] ;
   else
      time = TimeArrival[1] ;

2. if( TimeMachinesEnd[pool[thread_idx].permutation[0]] > minima )
   {
      nbTimes++ ;
      minima = TimeMachinesEnd[pool[thread_idx].permutation[0]];
   }

3. for(int k = 0 ; k < pool[thread_idx].limit1; k++)
      jobTime = jobEnd[k] ;
```

Consider the first "if" scenario. Let us suppose the values of LIMIT of the first 31 threads of a warp are not null except one. When that warp encounters the conditional instruction "if", only one thread passes through the condition checking and performs the assignment instruction. All the other 31 threads will be idle waiting for the thread 32 to be completed. The big deal in this case is that no other warps are allowed to run on that multiprocessor meanwhile because the warp is not completely idle. The same problem is encountered with the "for" loop. Suppose LIMIT1 of the first 31 threads of a warp is null while its value for the thread 32 is quite important. In that case, the 31 threads have to stay idle and wait until the other thread finishes its loop. The gap could be quite important since the value of LIMIT1 could be high depending on the size of the permutation being evaluated.

Some present techniques for handling branch divergence either demand hardware support [1] or require host-GPU interaction [11], which incurs overhead. Some other works such as [3] intervene at the code level. They expose a branch distribution method that aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths. In our work, we have also opted for software-based optimizations like [3]. In fact, we figure out how to literally rewrite the branching instructions into basic ones in order to make thread execution paths uniform. We also demonstrate that we could ameliorate performances only by judiciously reordering data being assigned to each thread.

4 Thread-Data Reordering

As explained in Section 2, any flow control instruction (if, switch, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge. If this happens, the different paths are executed in a serial way, increasing the total amount of instructions executed for this warp. It is important here to note that the threads execution path are data-dependent that is the data input set of a thread determines its behavior in a given kernel. Starting from this observation, we propose to reorder the data sets that the GPU threads work on.

The purpose of thread-data reordering is essentially to find an appropriate mapping between threads and input sets. In our work, we propose a reordering based on the data of the thread rather than its identifier like it is usually done [11]. Indeed, since the data of a given sub problem depend on its level in the search tree, the idea is to generate the pool of nodes to be evaluated in parallel from the same level unless from close levels on the tree. To do so we used the breadth-first exploration strategy (BFS) to generate the pool. Breadth first exploration expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. Initially the pool contains just the root. At each iteration, the node at the head of the pool is expanded. The generated nodes are then added to the tail of the pool. Using breadth-first branching guarantees that nodes belonging to the same level in the tree have much in common than other nodes generated by other decomposition paradigms namely depth first (DFS) or best first (BEFS) branching. Particularly in our case, nodes generated from the same father node have the same LIMIT1 and LIMIT2 but have different jobs to schedule.

To evaluate the performance of the proposed approach we run experiments over Flow-shop instances proposed by Taillard in [10]. Taillard's benchmarks are the best known problem instances for basic model with makespan objective. Each Taillard's instance NxM defines the number N of jobs to be scheduled and the number of machines M on which the jobs are scheduled. For each experiment, different pool sizes and problem instances are considered. The approach has been implemented using C-CUDA 4.0. The experiments have been carried out using a an Intel Xeon E5520 bi-processor coupled with a GPU device. The bi-processor is 64-bit, quad-core and has a clock speed of 2.27GHz. The GPU device is an

Nvidia Tesla C2050 with 448 CUDA cores (14 multiprocessors with 32 cores each) and a 2.8GB global memory.

Pool Size \ Instances		20x20	50x20	100x20	200x20	500x20
4096	DFS	191.1	214.41	242.28	283.4	383.84
16x256	BFS	186.75	204.58	238.1	275.43	366.47
8192	DFS	198.73	218.53	248.39	293.69	408.2
32x256	BFS	194.22	209.26	241.46	287.18	404.72
16384	DFS	224.82	253.36	300.21	351.14	546.62
64x256	BFS	216.86	235.78	291.10	325.93	521.80
32768	DFS	231.02	276.91	340.15	426.98	761.38
128x256	BFS	219.5	259.16	316.5	410.52	711.86
65536	DFS	269.8	318.52	405.8	568.76	1133.97
256x256	BFS	253.96	301.81	380.95	543.84	1090.22

Table 1. Time measurements on GPU without data reordering (using DFS) and with data reordering (using BFS)

The obtained results are reported in Table 1. Each pool size in the first column is expressed as bxt where b and t designates respectively the number of blocks and the number of threads within a block. Each node in the pool is evaluated by exactly one thread. We notice that although the speed-up is not impressive, reordering data makes the kernel run faster than with a pool generated via a DFS exploration strategy. For a same pool size, the acceleration grows accordingly with the permutation size associated to the instance. For example, in the instance corresponding to the scheduling of 500 jobs over 20 machines, the permutation size is equal to 500. For this instance, a pool of 4096 sub-problems would contain nodes from the same level of the exploration tree whereas the same pool generated in instance 20 jobs over 20 machines would have sub-problems from different levels. This explains why the acceleration follows the permutation size behavior. Generating the pool to evaluate using a breadth first strategy guarantees for large instances to have nodes from almost the same level. This allows to reduce the impact of conditional instructions that depends of the values of LIMIT1 and LIMIT2.

5 Branch refactoring

To reduce the divergence caused by conditional instructions, we use the branch refactoring approach. This latter consists in rewriting the conditional instructions so that threads of the same warp execute an uniform code avoiding their divergence. To do that, we study in the following different “if” scenarios and propose some optimizations accordingly. Consider a generalization of the if-else statement of the scenario (1) exposed in the Section 3. In this case, the conditional expression compares the content of a variable to 0. The idea is to replace

this conditional expression by two functions namely f and g as explained in the Equation 1.

$$\begin{array}{ll} if(x \neq 0) & if(x \neq 0) \\ \quad a = b[1]; & \quad a = b[1] + 0 \times c[1]; \\ else & else \\ \quad a = c[1]; \quad \Rightarrow & \quad a = 0 \times b[1] + c[1]; \end{array}$$

$$\Rightarrow a = f(x) \times b[1] + g(x) \times c[1];$$

$$where: \ \text{f(x)}=\begin{cases} f(x) = 0 \ if \quad x = 0 \\ 1 \qquad\quad else \end{cases} \quad and \quad \text{g(x)}=\begin{cases} g(x) = 1 \ if \quad x = 0 \\ 0 \qquad\quad else \end{cases} \tag{1}$$

The behavior of f and g fits the trigonometric function cosinus. This function returns values between 0 and 1. Particularly, we defined an integer variable to which we assign the result of the cosinus function as quoted in the above code. The value taken would only be 0 or 1 since it would be rounded to 0 if it is not equal to 1. In order to increase performance we used CUDA runtime math operations: __sinf(x), __expf(x) and so forth. Those functions are mapped directly to the hardware level [8]. They are faster but provide lower accuracy which does not matter in our case because we do round results to int. The throughput of __sinf(x), __cosf(x), __expf(x) is 1 operation per clock cycle [8].

Result of branch rewriting for the scenario (1)

```
int coeff = __cosf(pool[tid].limit1);
time = (1 - coeff) * TimeMachines[1] + coeff * TimeArrival[1];
```

Let us now consider a scenario with an "if" statement which compares two values between themselves like shown in Equation 2.

$$\begin{array}{lll} & if(x \gtrsim y) \quad a = b[1]; & \Rightarrow \quad if(x - y \geq 1) \quad a = b[1]; \\ \Rightarrow & if(x - y - 1 \geq 0) \qquad a = b[1]; & (x, y) \in N \\ \Rightarrow & a = f(x, y) \times b[1] + g(x, y) \times a; & \end{array} \tag{2}$$

$$where: \qquad \text{f(x,y)}=\begin{cases} 1 \ if \ x - y - 1 \geq 0 \\ 0 \ if \ x - y - 1 < 0 \end{cases}$$

$$and \qquad \text{g(x,y)}=\begin{cases} 0 \ if \ x - y - 1 \geq 0 \\ 1 \ if \ x - y - 1 < 0 \end{cases}$$

We do the same transformations than before using the exponential function. The exponential is a positive function which is equal to 1 when applied to 0.

Thus, if x is less than y __expf(x-y-1) returns a value between 0 and 1. If we round this result to an integer value we obtain 0. Now, if x is greater than y __expf(x-y-1) return a value greater than 1 and since we get the minimum between 1 and the exponential, the returned result would be 1. This behavior exactly satisfies our prerequisites. The "if" instruction is now equivalent to:

```
int coeff = min(1, __expf(x - y - 1));
a = coeff * b[1] + ( 1 - coeff ) * a ;
```

The effectiveness of both transformations was tested on the same configuration used in Section 4. Table 2 compares the parallel efficency of the GPU-based evaluation of bounds with and without using code optimizations. The reported speed ups are calculated relatively to the sequentiel version considering the ratio between the measured execution times.

Pool Size \ Instances		20x20	50x20	100x20	200x20	500x20
4096	refactored	1.17	2.14	3.24	6.93	10.24
16x256	basic	1.05	1.89	3.06	5.17	9.66
8192	refactored	2.25	4.20	6.35	10.77	18.44
32x256	basic	2.01	3.72	5.99	9.88	17.39
16384	refactored	4.00	7.96	10.52	18.92	28.55
64x256	basic	3.58	7.04	9.92	17.36	26.93
32768	refactored	7.99	11.90	19.52	30.02	41.76
128x256	basic	7.13	10.53	18.42	27.54	39.40
65536	refactored	10.43	15.15	32.38	45.76	54.53
256x256	basic	9.31	13.41	30.55	41.98	51.44

Table 2. Parallel speedup obtained with/without code optimization

The reported accelerations no doubtly prove that bound evaluation parallelization on top of GPU provides an efficient way for speed up B&B algorithms. In fact, the GPU-based evaluation runs up to 51.44 times faster than the CPU one. The other important result, is that using thread divergence reduction improves the classic GPU acceleration. This acceleration improvement grows acccordingly to the size of the problem instance and the size of the pool of sub-problems considered in the experiment. Indeed, with a pool of 65536 nodes and an instance of 500 jobs and 20 machines a speed up of x54,5 is achieved while it reaches only x10,2 with a pool of 4096 nodes.

6 Conclusion and future work

In this work, we have investigated using GPUs to improve the performance of B&B algorithms. To the best of our knowledge, no contribution has been

proposed to raise such challenge. We focused on the parallel evaluation of the bounds for the Flow-shop permutation problem. In order to face out irregularities caused by data dependent branching and leading to thread divergence, we have proposed some software based optimizations. Experiments reported that parallel evaluation of bounds speed up executions up to 54.5 times compared to a CPU version. This promising results could be easily improved when the approach is combined with an optimized data access to GPU memory spaces.

As future contribution, we aim to focus on memory management issues related to data inputs for combinatorial optimization problems. Indeed, when working with such structures usually large in size many memory transactions are performed leading to a global loss of performance. Our direction for future work is also to generate the pool of subproblems directly on GPU. This modification would reduce the transfer time of data structures from CPU to GPU. The challenging issue of this approach is to find an efficient mapping between a thread id and the nodes to generate.

References

1. W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 407-420, Washington, DC, USA, 2007.
2. B. Gendron and T.G. Crainic. Parallel Branch and Bound Algorithms: Survey and Synthesis. Operations Research, 42:10421066, 1994.
3. T.Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4). ACM, New York, USA, Article 3 , 8 pages. 2011.
4. B. Jang and et al. Exploiting memory access patterns to improve memory performance in data-parallel architectures. IEEE Trans. on Parallel and Distributed Systems, 22(1):105 118, Jan. 2011.
5. S.M. Johnson. Optimal two and three-stage production schedules with setup times included. Naval Research Logistis Quarterly, 1:6168, 1954.
6. J.K.Lenstra, B.J.Lageweg, and A.H.G.Rinnooy Kan. A General bounding scheme for the permutation Flow-shop problem. Operations Research, 26(1):5367, 1978.
7. N. Melab. Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul. HDR thesis, LIFL, USTL, Novembre 2005.
8. NVIDIA CUDA C Programming Best Practices Guide. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`
9. S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. W. Hwu. Program optimization carving for gpu computing. J.Parallel Distributed Computing, 68(10):13891401,2008.
10. E.Taillard. Benchmarks for basic scheduling problems. European Journal of European Research, pages 23:661673, 1993.
11. Eddy Z. Zhang, Y. Jiang, Z.Guo, and X.Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10). ACM, New York, NY, USA, 115-126. 2010.